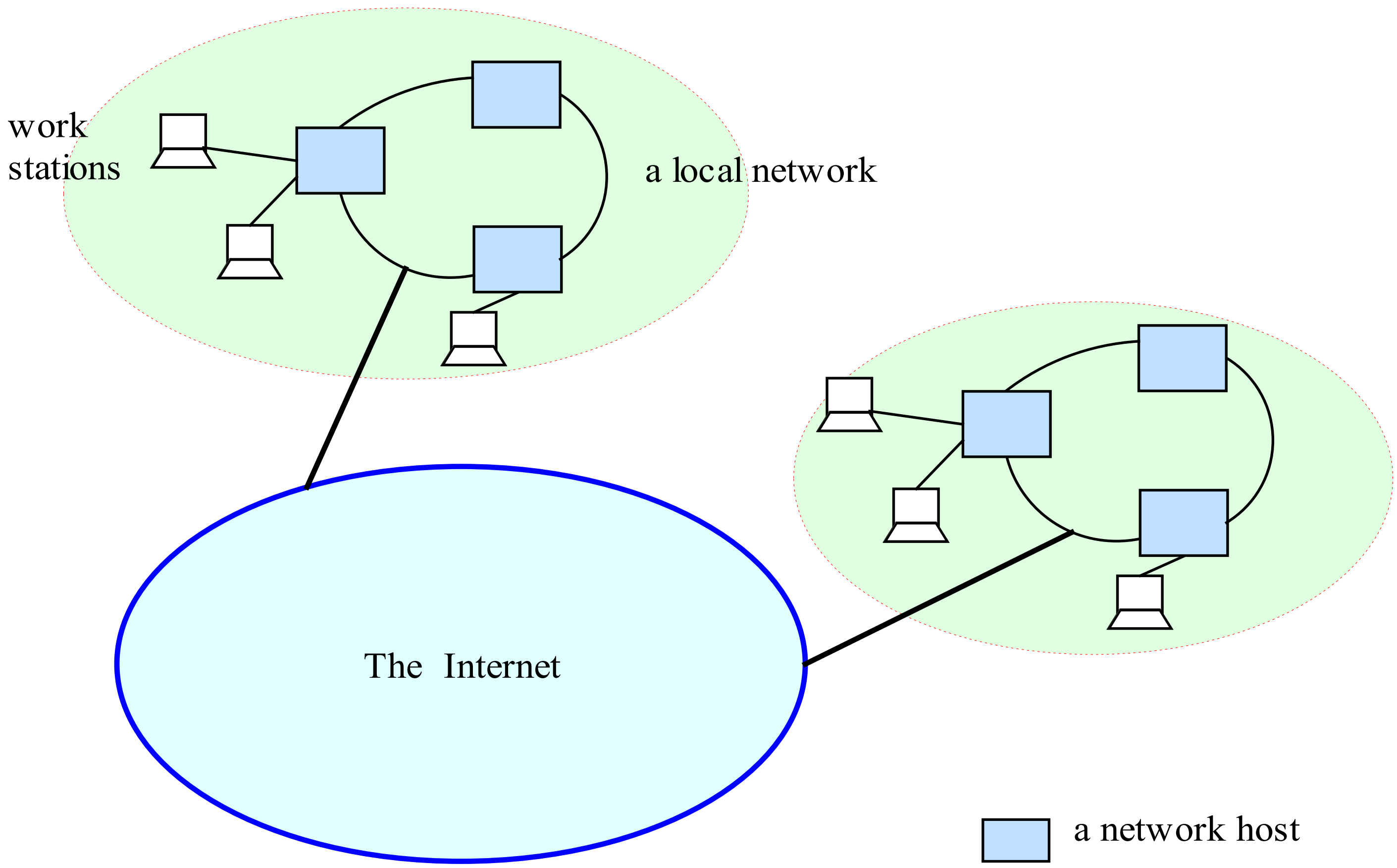
# COMP 6231: Distributed System Design

## Network and Process Communication

Based on Chapters 3 and 4 of the text book and the slides from Prof. M.L. Liu, California Polytechnic State University

# Distributed Systems



# Interprocess Communications

- Distributed computing requires information to be exchanged among independent processes.
- Operating systems provide facilities for interprocess communications (IPC), such as message queues, semaphores, and shared memory.
- Distributed computing systems make use of these facilities to provide application programming interface which allows IPC to be programmed at a higher level of abstraction.

# Some terminology

- A message is used to communicate between processes. Arbitrary size byte stream.
- A packet is a fragment of a message that might travel on the wire. Variable size but limited, usually to 1400 bytes or less.
- A protocol is an algorithm by which processes cooperate to do something using message exchanges.
- A network is the infrastructure that links the computers, workstations, terminals, servers, etc.
  - It consists of routers
  - They are connected by communication links

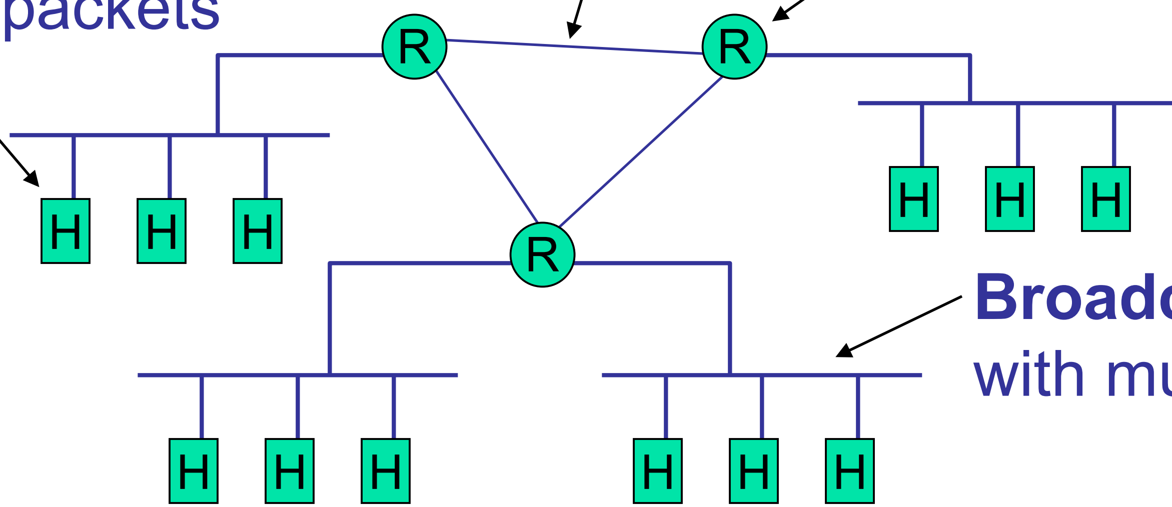
# Network components

**Point to point link**: link with two nodes (router or host)

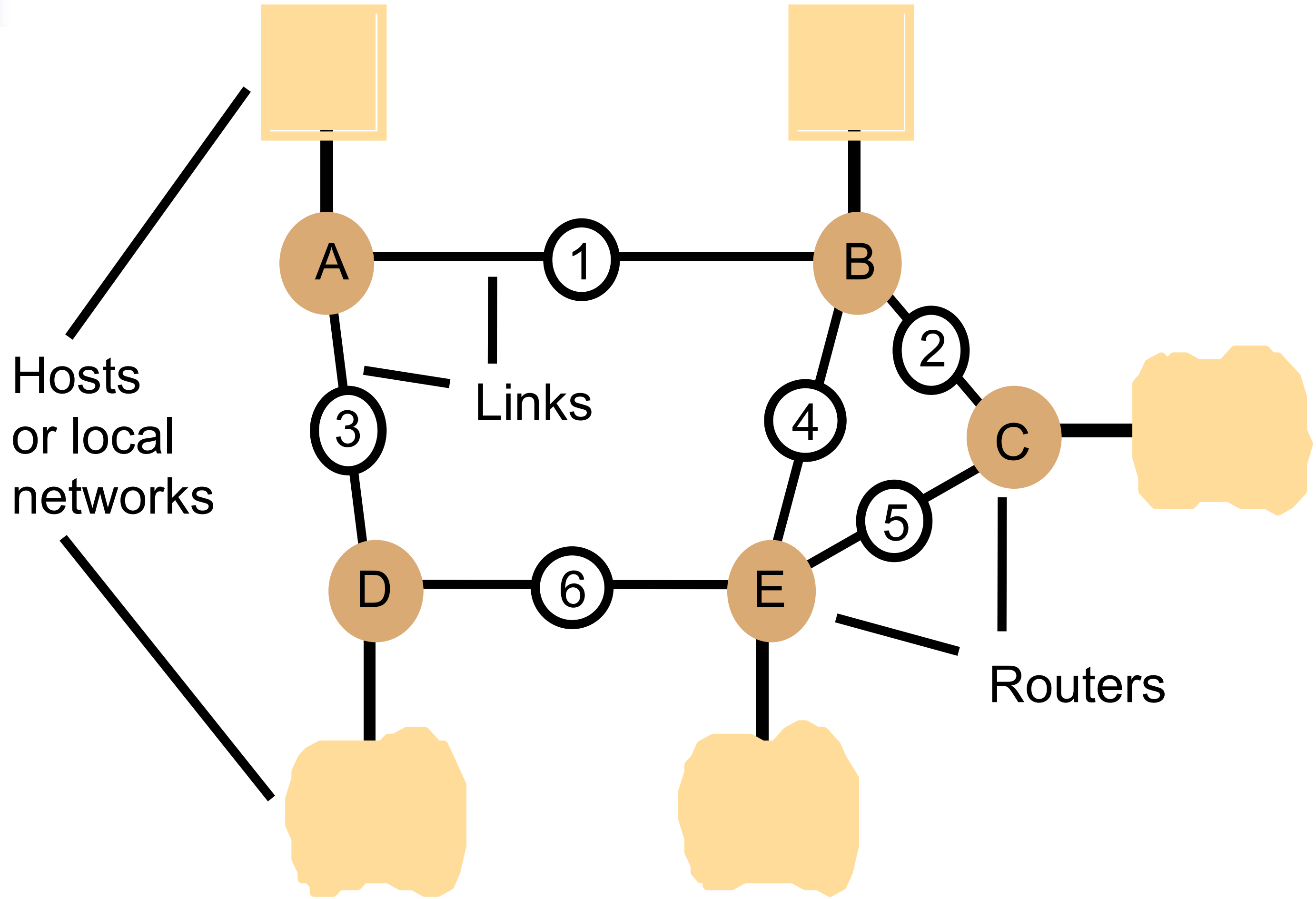
**Router**: Forwards IP packets

**Host**: Source and sink of IP packets

**Broadcast link**: link with multiple nodes



# Routing in a wide area network



# Routing tables for the network

| Routings from A | | |
|---|---|---|
| *To* | *Link* | *Cost* |
| A | local | 0 |
| B | 1 | 1 |
| C | 1 | 2 |
| D | 3 | 1 |
| E | 1 | 2 |

| Routings from B | | |
|---|---|---|
| *To* | *Link* | *Cost* |
| A | 1 | 1 |
| B | local | 0 |
| C | 2 | 1 |
| D | 1 | 2 |
| E | 4 | 1 |

| Routings from C | | |
|---|---|---|
| *To* | *Link* | *Cost* |
| A | 2 | 2 |
| B | 2 | 1 |
| C | local | 0 |
| D | 5 | 2 |
| E | 5 | 1 |

| Routings from D | | |
|---|---|---|
| *To* | *Link* | *Cost* |
| A | 3 | 1 |
| B | 3 | 2 |
| C | 6 | 2 |
| D | local | 0 |
| E | 6 | 1 |

| Routings from E | | |
|---|---|---|
| *To* | *Link* | *Cost* |
| A | 4 | 2 |
| B | 4 | 1 |
| C | 5 | 1 |
| D | 6 | 1 |
| E | local | 0 |

# Networks are not reliable!

- Different packets/messages between a sender and a receiver may take different paths/time and arrive out of order.
- Links can corrupt messages
  - Rare in the high quality ones on the Internet “backbone”
  - More common with wireless connections, cable modems, ADSL
- Routers can get overloaded
  - When this happens they drop messages
  - This is very common
- But protocols that retransmit lost packets can increase reliability

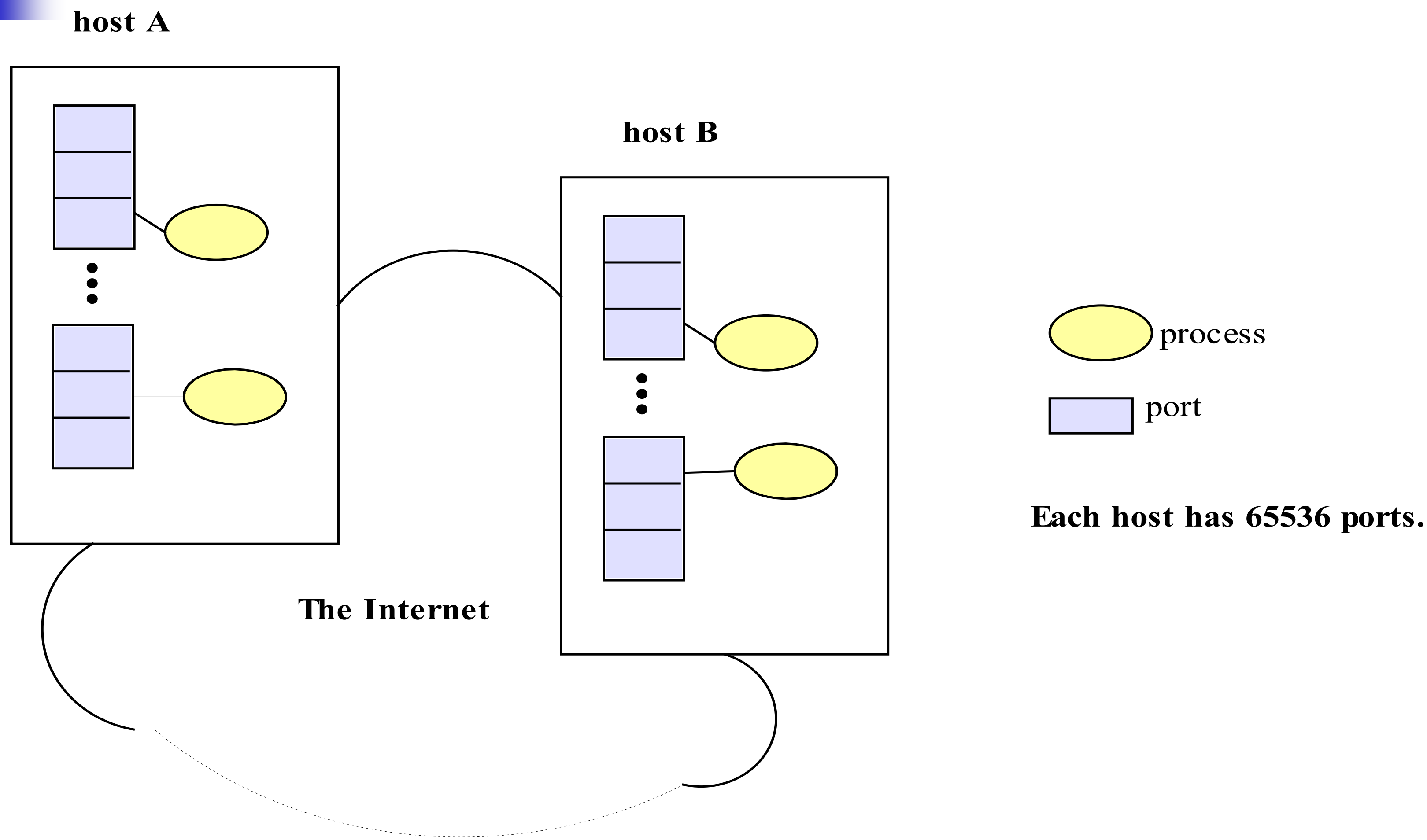
# IP (Internet Protocol)

- Three services:
  - *Unicast*: transmits a packet to a specific host
  - *Multicast*: transmits a packet to a group of hosts
  - *Anycast*: transmits a packet to one of a group of hosts (typically nearest)
- Destination and source identified by the IP address (32 bits for IPv4, 128 bits for IPv6)
- All services are unreliable
  - Packet may be dropped, duplicated, and received in a different order

# IP addresses (v4)

- In binary, a 32-bit integer
- In text, this:  “128.52.7.243”
  - Each decimal digit represents 8 bits (0 – 255)
- “Private” addresses are not globally unique:
  - Used behind NAT boxes
  - 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16
- Multicast addresses start with 1110 as the first 4 bits (Class D address)
  - 224.0.0.0/4
- Unicast and anycast addresses come from the same space

# Logical Ports



# Well-known ports

## Assignment of some well-known ports

| Protocol | Port | Service |
|---|---|---|
| echo | 7 | IPC testing |
| daytime | 13 | provides the current date and time |
| ftp | 21 | file transfer protocol |
| telnet | 23 | remote, command-line terminal session |
| smtp | 25 | simple mail transfer protocol |
| time | 37 | provides a standard time |
| finger | 79 | provides information about a user |
| http | 80 | web server |
| RMI Registry | 1099 | registry for Remote Method Invocation |
| special web server | 8080 | web server which supports servlets, JSP, or ASP |

# UDP (User Datagram Protocol)

- Runs above IP
- Same unreliable service as IP
  - Packets can get lost anywhere:
    - Outgoing buffer at source
    - Router or link
    - Incoming buffer at destination
  - Messages may be delivered out of send order
- But adds port numbers
  - Used to identify “application layer” protocols or processes
- Also a checksum, optional

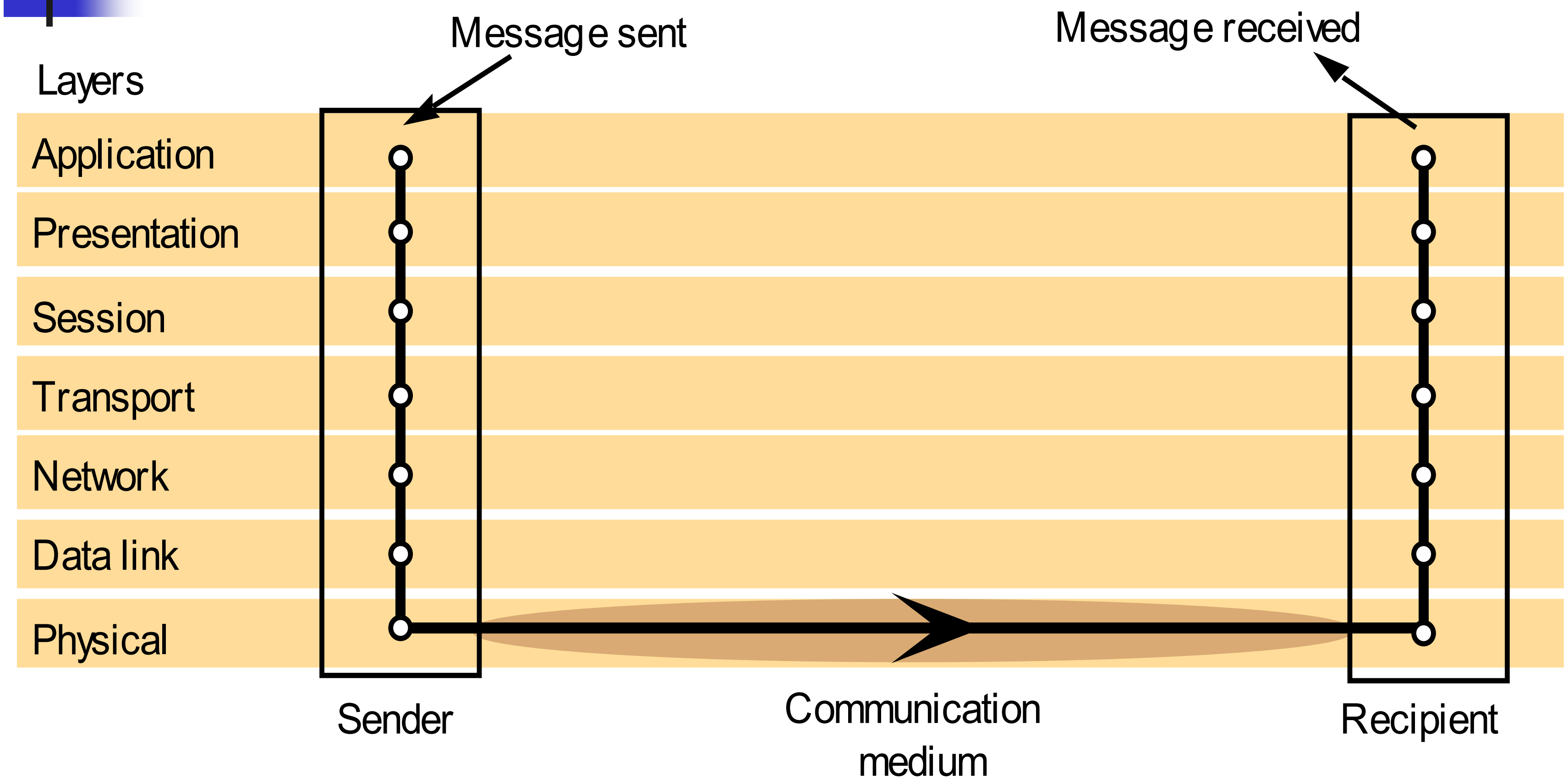
# TCP (Transmission Control Protocol)

- Runs above IP
  - Port number and checksum like UDP
- Service is in-order byte stream
  - Application does not absolutely know how the bytes are packaged in packets
- Flow control and congestion control
- Connection setup and teardown phases
- Can be considerable delay between bytes out at source and bytes in at destination
  - Because of timeouts and retransmissions
- Works only with unicast (not multicast or anycast)

# UDP vs. TCP

- UDP is more real-time
  - Packet is sent or dropped, but is not delayed
- UDP has more of a "message" flavor
  - One packet = one message
  - But must add reliability mechanisms over it
- TCP is great for transferring a file or a bunch of email, but kind-of frustrating for messaging
  - Interrupts to application don't conform to message boundaries
  - No "Application Layer Framing"
- TCP is vulnerable to DoS (Denial of Service) attacks, because initial packet consumes resources at the receiver

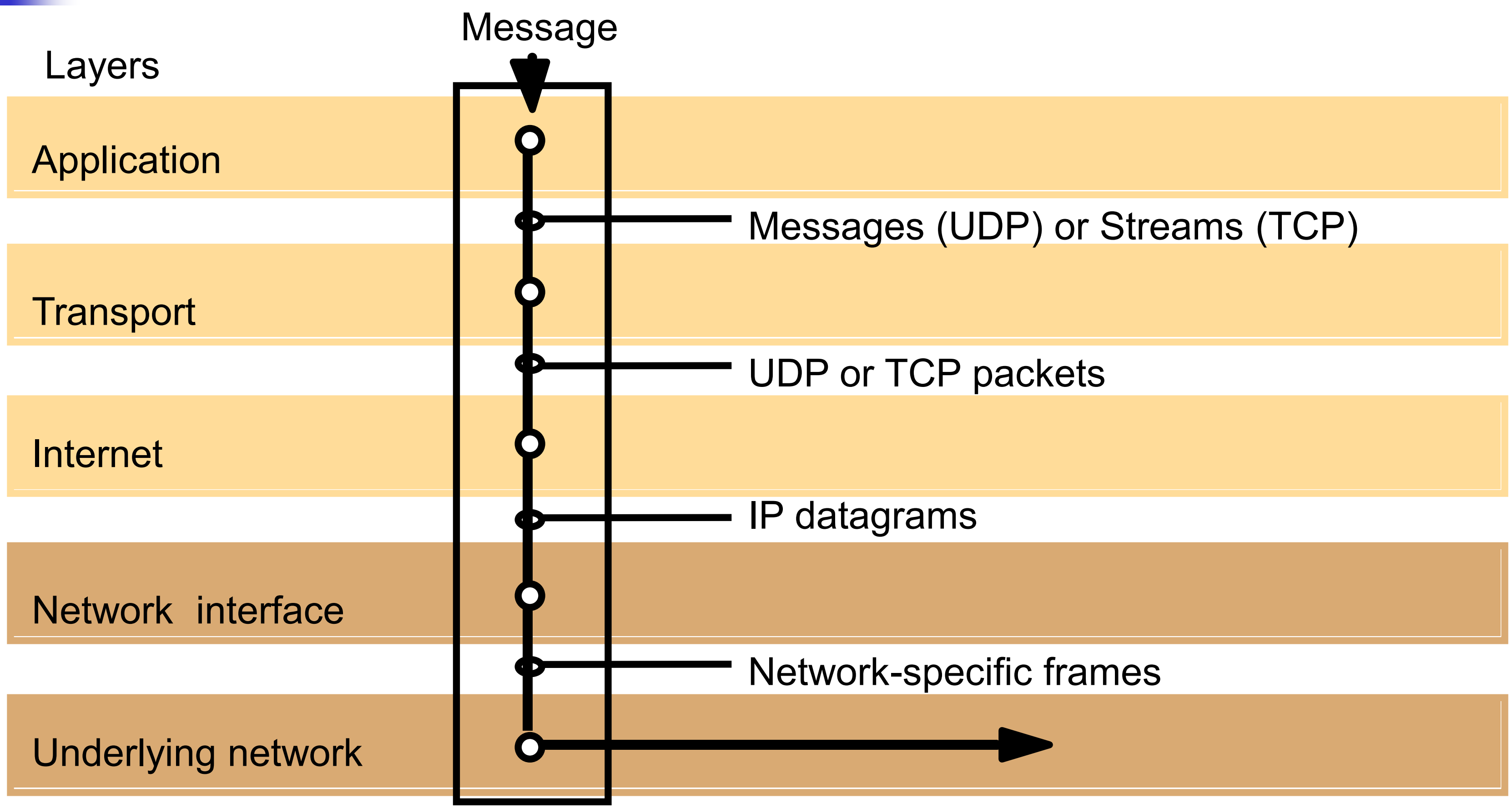
# Classic OSI stack



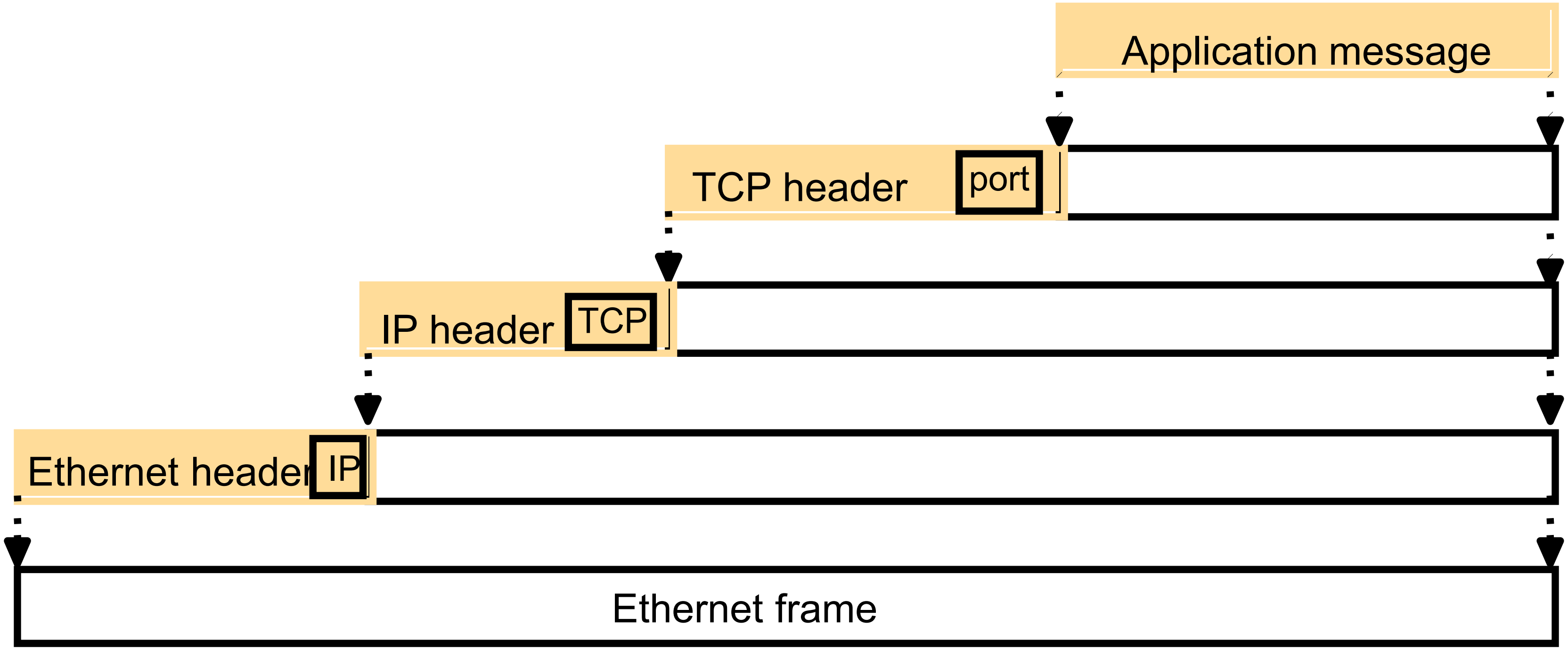
# OSI protocol summary

| *Layer* | *Description* | *Examples* |
|---|---|---|
| Application | Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service. | HTTP, FTP , SMTP, CORBA IIOP |
| Presentation | Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers, which may differ. Encryption is also performed in this layer, if required. | Secure Sockets (SSL),CORBA Data Representation |
| Session | At this level reliability and adaptation are performed, such as detection of failures and automatic recovery. | |
| Transport | This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes, Protocols in this layer may be connection-oriented or connectionless. | TCP, UDP |
| Network | Transfers data packets between computers in a specific network. In a WAN or an internetwork this involves the generation of a route passing through routers. In a single LAN no routing is required. | IP, ATM virtual circuits |
| Data link | Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN transmission is between pairs of routers or between routers and hosts. In a LAN it is between any pair of hosts. | Ethernet MAC, ATM cell transfer, PPP |
| Physical | The circuits and hardware that drive the network. It transmits sequences of binary data by analogue signalling, using amplitude or frequency modulation of electrical signals (on cable circuits), light signals (on fibre optic circuits) or other electromagnetic signals (on radio and microwave circuits). | Ethernet base-band signalling, ISDN |

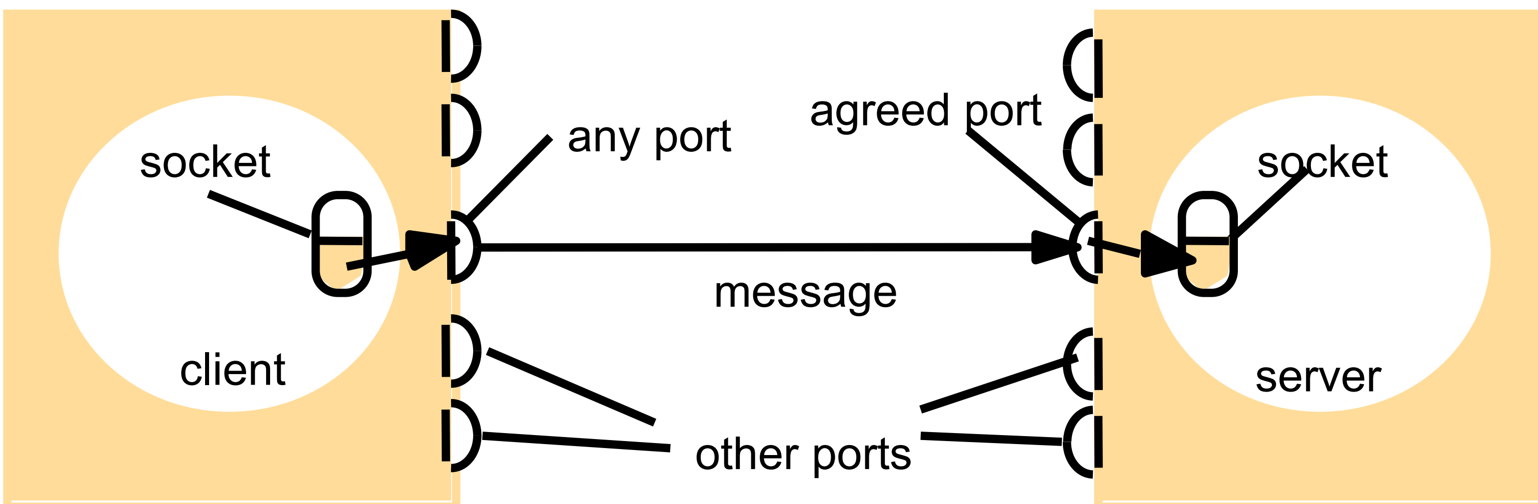
# TCP/IP layers



# Encapsulation in a TCP/IP message



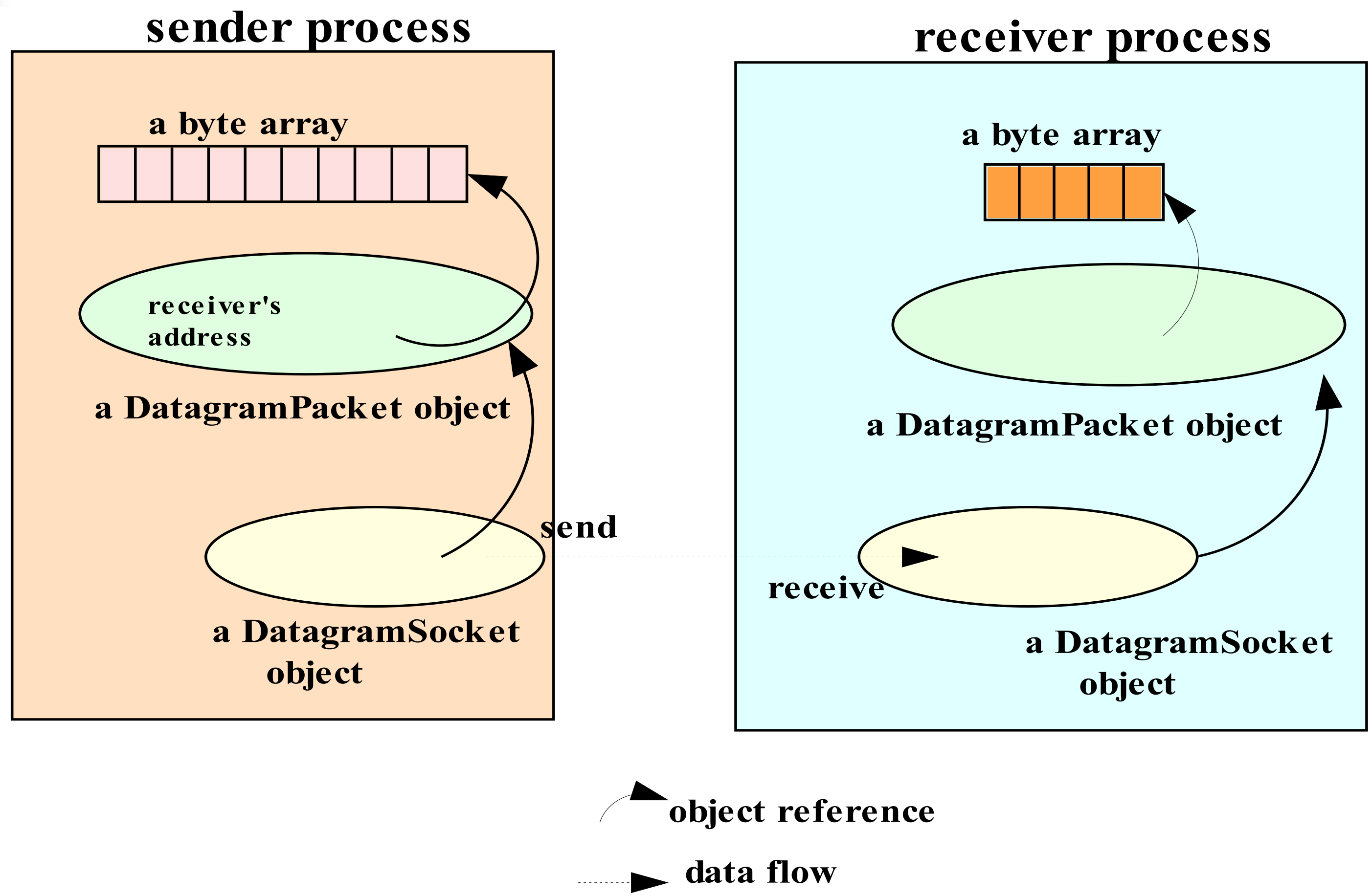
# Sockets and ports




Internet address = 138.37.94.248

Internet address = 138.37.88.249

# The Data Structures in the sender and receiver programs



# The program flow in the sender and receiver programs

**sender program**

create a datagram socket and
bind it to any local port;
place data in a byte array;
create a datagram packet, specifying
the data array and the receiver's
address;
invoke the send method of the
socket with a reference to the
datagram packet;

**receiver program**

create a datagram socket and
bind it to a specific local port;
create a byte array for receiving the data;
create a datagram packet, specifying
the data array;
invoke the receive method of the
socket with a reference to the
datagram packet;

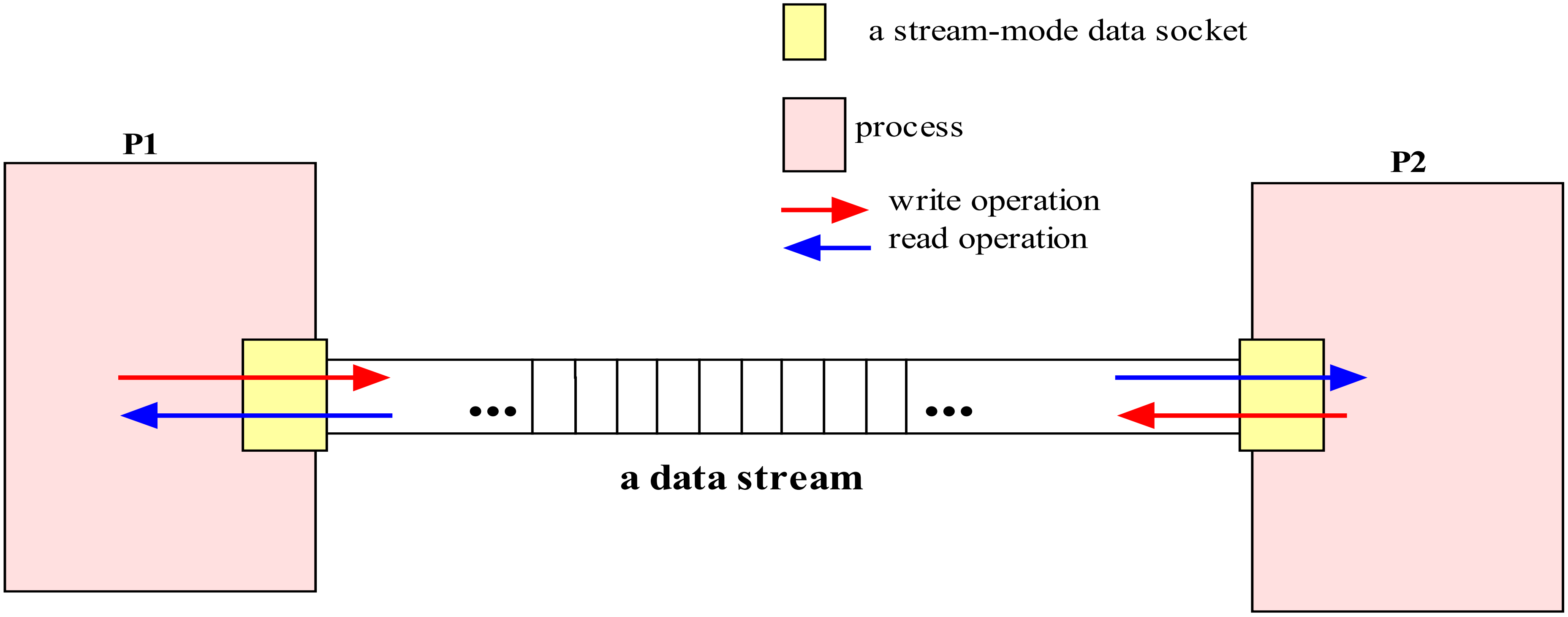
# UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
            DatagramSocket aSocket = null;
             try {
                    aSocket = new DatagramSocket();
                    byte [] m = args[0].getBytes();
                    InetAddress aHost = InetAddress.getByName(args[1]);
                    int serverPort = 6789;
                    DatagramPacket request =
                            new DatagramPacket(m, args[0].length(), aHost, serverPort);
                    aSocket.send(request);
                    byte[] buffer = new byte[1000];
                    DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                    aSocket.receive(reply);
                    System.out.println("Reply: " + new String(reply.getData()));
             }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
             }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(aSocket != null) aSocket.close();} } }
```

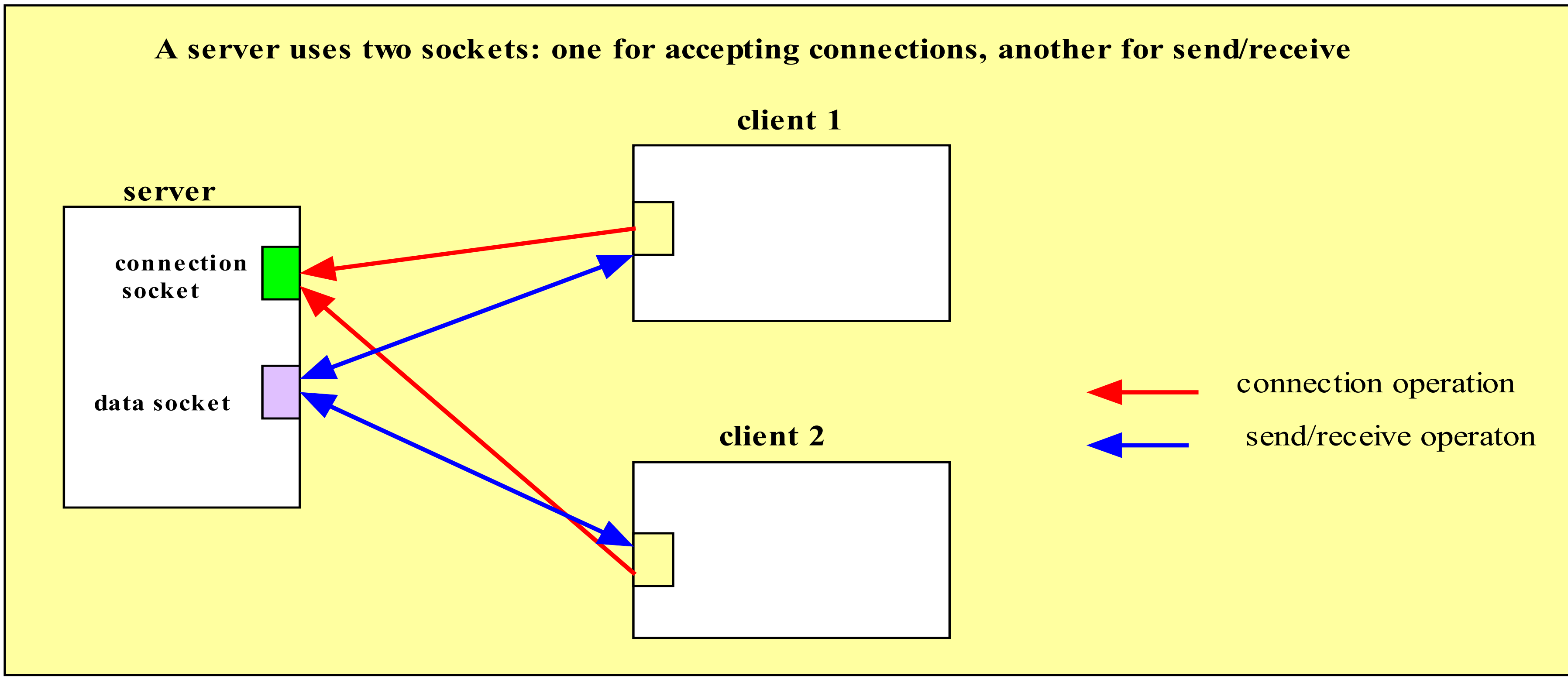
# UDP echo server

```
import java.net.*;
import java.io.*;
public class UDPServer{
		public static void main(String args[]){
		DatagramSocket aSocket = null;
		   try{
				aSocket = new DatagramSocket(6789);
				byte[] buffer = new byte[1000];
				while(true){
				   DatagramPacket request = new DatagramPacket(buffer, buffer.length);
				  aSocket.receive(request);
				  DatagramPacket reply = new DatagramPacket(request.getData(),
					request.getLength(), request.getAddress(), request.getPort());
				  aSocket.send(reply);
				}
		   }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
		  }catch (IOException e) {System.out.println("IO: " + e.getMessage());;}
		}finally {if(aSocket != null) aSocket.close();;}
    }
}
```

# Stream-mode Socket API (connection-oriented)



# The server (the connection listener)



# Stream-mode Socket API program flow

**connection listener (server)**

create a connection socket
and listen for connection
requests;
accept a connection;
creates a data socket for reading from
or writing to the socket stream;
get an input stream for reading
to the socket;
read from the stream;
get an output stream for writing
to the socket;
write to the stream;
close the data socket;
close the connection socket.

**connection requester (server)**

create a data socket
and request for a connection;

get an output stream for writing
to the socket;
write to the stream;

get an input stream for reading
to the socket;
read from the stream;
close the data socket.

# TCP client sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
            try{
                int serverPort = 7896;
                s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out = new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);          // UTF is a string encoding see Sn 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
            }catch (UnknownHostException e){System.out.println("Sock:"+e.getMessage());
            }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
            }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException e)
                                        {System.out.println("close:"+e.getMessage());} } } }
```

# TCP echo server

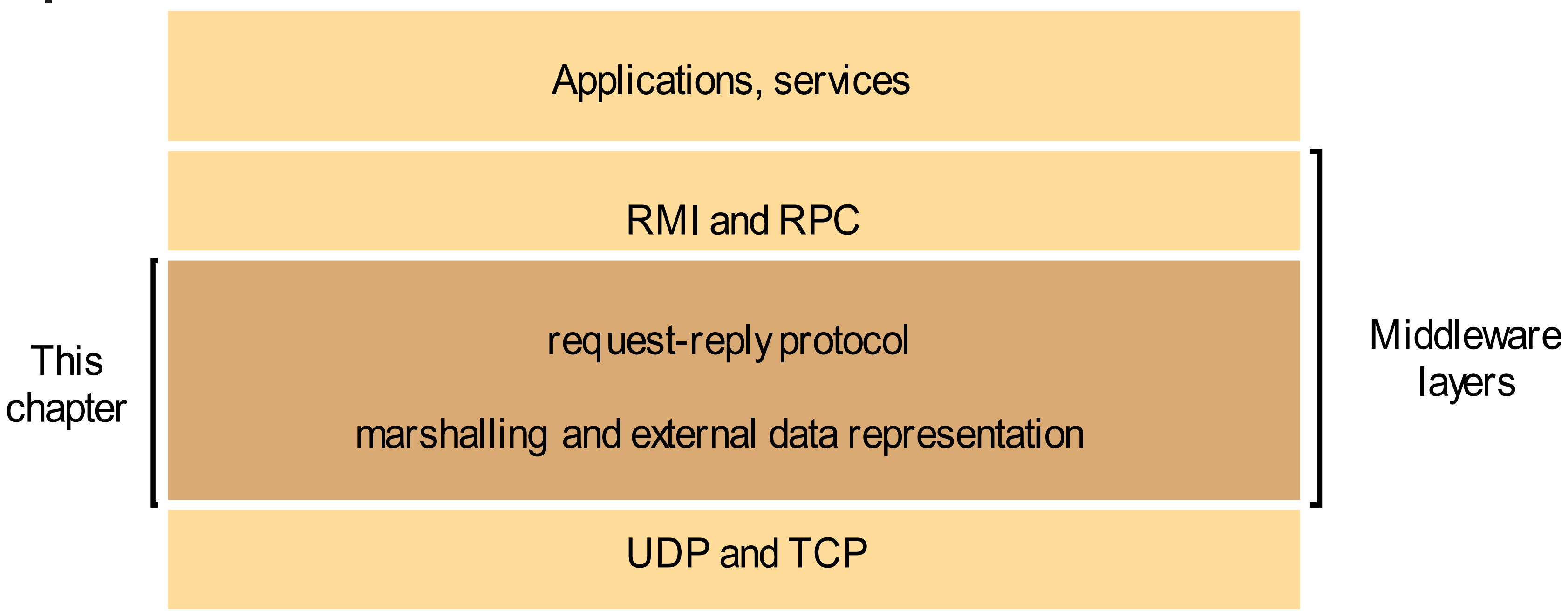
```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
            try{
                    int serverPort = 7896;
                    ServerSocket listenSocket = new ServerSocket(serverPort);
                    while(true) {
                            Socket clientSocket = listenSocket.accept();
                            Connection c = new Connection(clientSocket);
                    }
            } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}

// this figure continues on the next slide
```

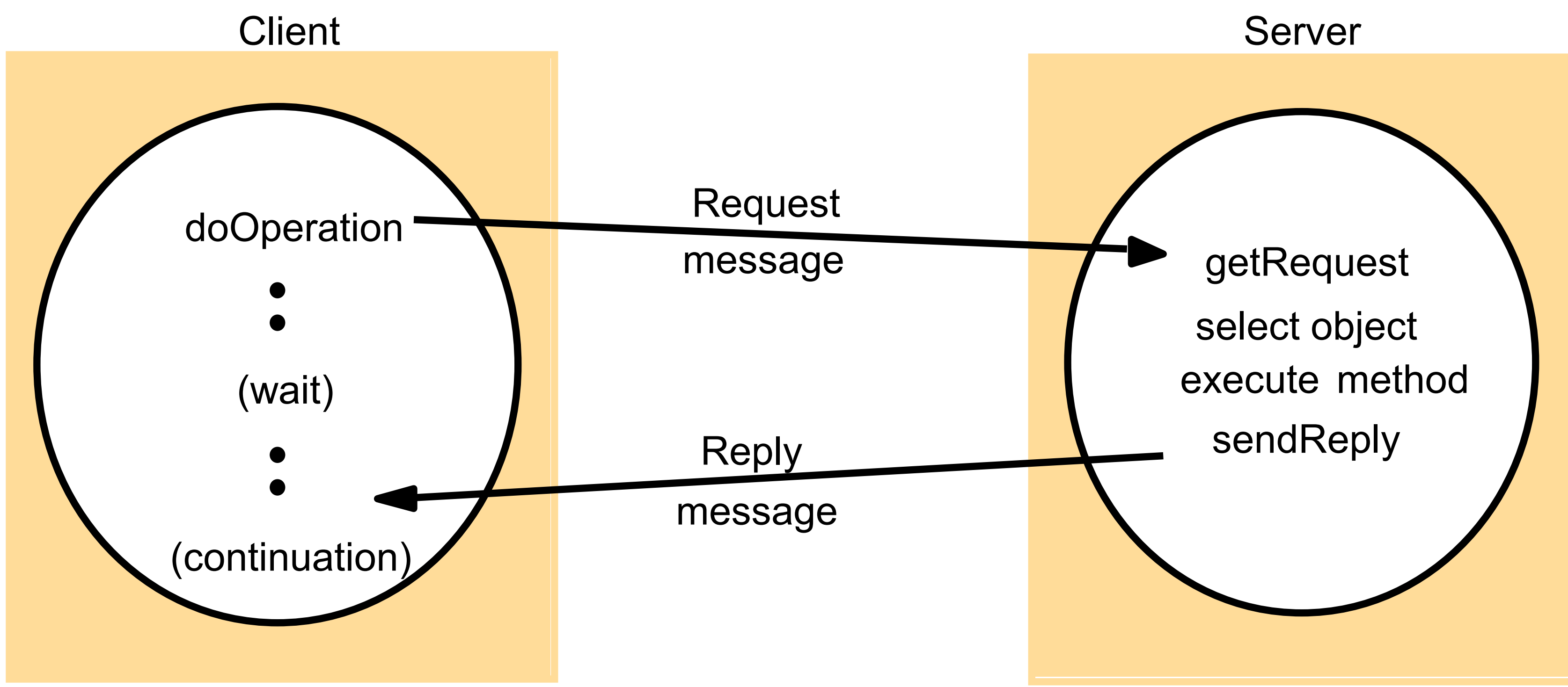
# TCP echo server continued

```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
          try {
                clientSocket = aClientSocket;
                in = new DataInputStream( clientSocket.getInputStream());
                out =new DataOutputStream( clientSocket.getOutputStream());
                this.start();
           } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
        }
        public void run(){
          try {                                     // an echo server
                String data = in.readUTF();
                out.writeUTF(data);
          } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
          } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
          } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}}}
```

# Remote Procedure Call using Sockets



# Request-reply communication



# Client-Server concept

- Server program is shared by many clients
- RPC protocol typically used to issue requests
- Server may manage special data, run on an especially fast platform, or have an especially large disk
- Client systems handle "front-end" processing and interaction with the human user

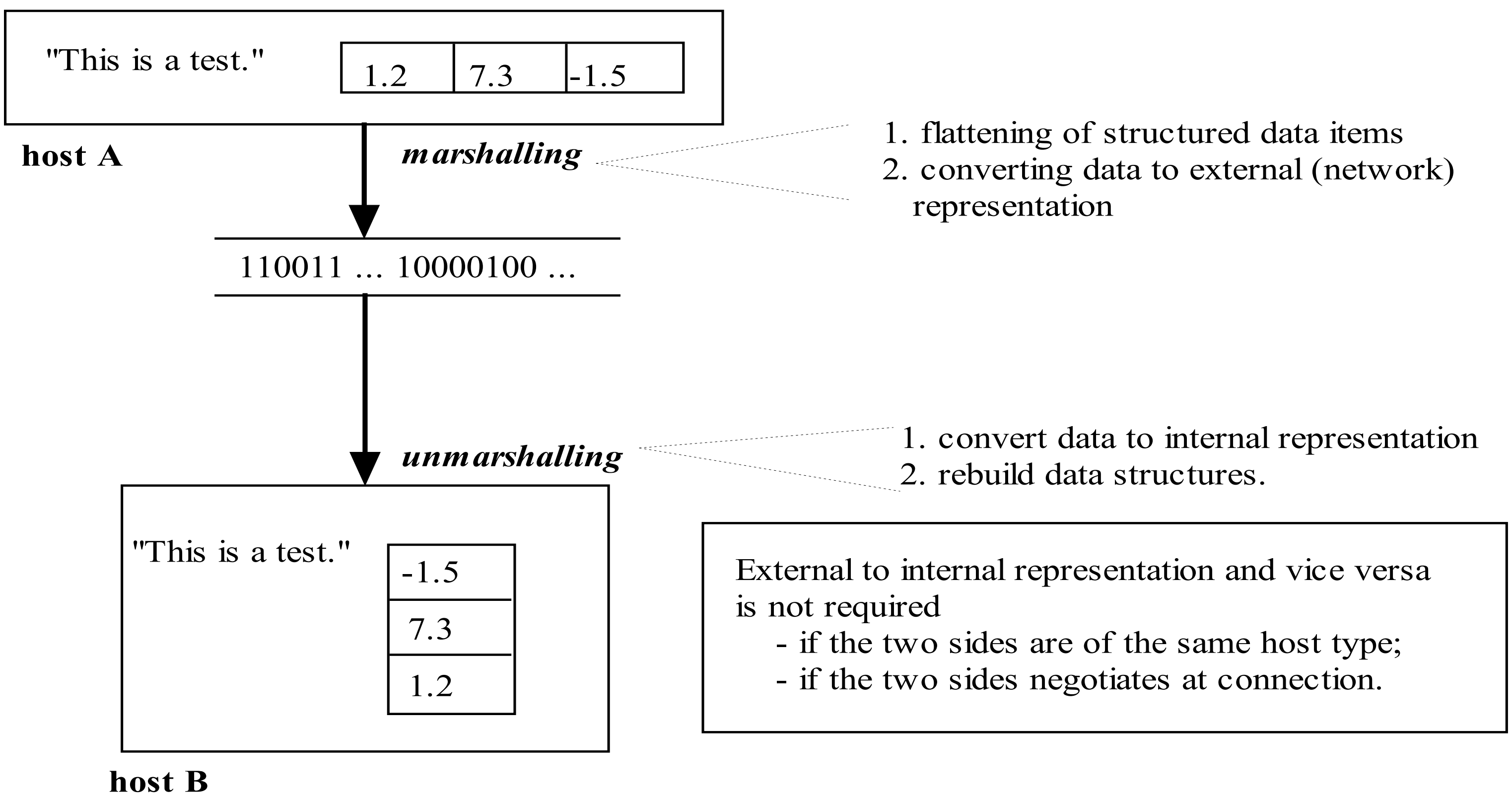
# Binding stage

- Occurs when client and server program first start execution
- Server registers its network address with name directory, perhaps with other information
- Client scans directory to find appropriate server
- Depending on how RPC protocol is implemented, may make a “connection” to the server, but this is not mandatory

# Data in messages

- We say that data is “marshalled” into a message and “unmarshalled” from it
- Representation needs to deal with byte ordering issues (big-endian versus little-endian), strings (some CPUs require padding), alignment, etc
- Goal is to be as fast as possible on the most common architectures, yet must also be very general

# Data Marshalling



# Request marshalling

- Client builds a message containing arguments, indicates what procedure to invoke
  - Due to the need for generality, data representation a potentially costly issue!
- Performs a send I/O operation to send the message
- Performs a receive I/O operation to accept the reply
- Unpacks the reply from the reply message
- Returns result to the client program

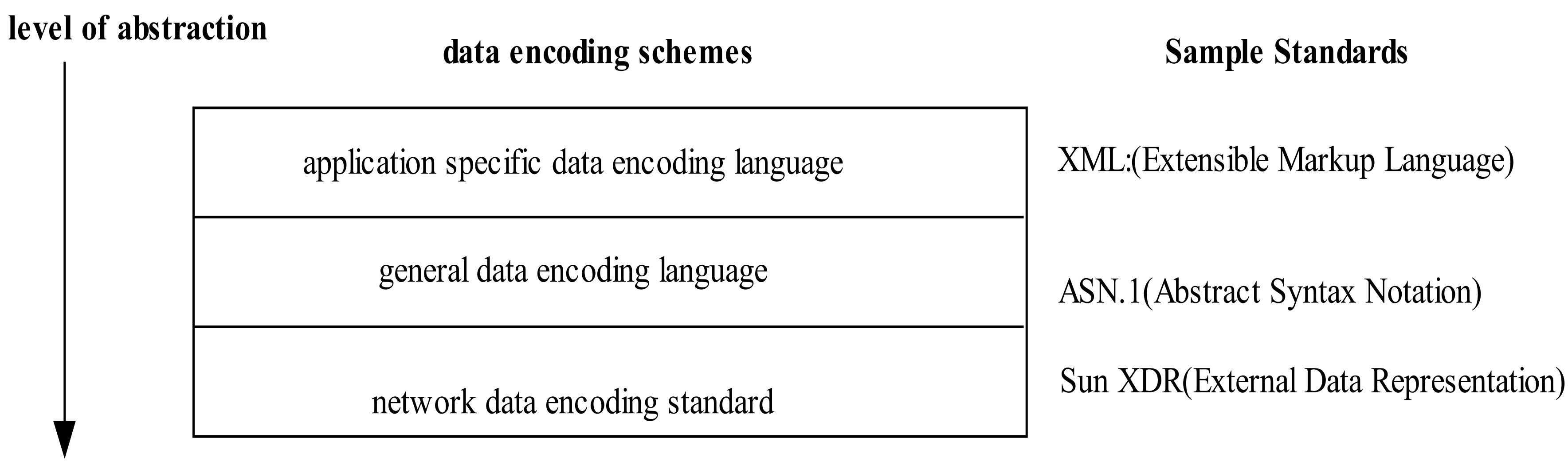
# Data Representation

- Data transmitted on the network is a binary stream.
- An interprocess communication system may provide the capability to allow data representation to be imposed on the raw data.
- Because different computers may have different internal storage format for the same data type, an external representation of data may be necessary.
- *Data marshalling* is the process of (i) flattening a data structure, and (ii) converting the data to an external representation.

# Data Encoding Protocols

Some well known external data representation schemes are:

- Sun XDR
- ASN.1 (Abstract Syntax Notation)
- XML (Extensible Markup Language)



# RPC "semantics"

- At most once: request is processed 0 or 1 times
- Exactly once: request is always processed 1 time
- At least once: request processed 1 or more times

*… but exactly once is impossible because we can't distinguish packet loss from true failures! In both cases, RPC protocol simply times out.*

# RPC versus local procedure call

- Restrictions on argument sizes and types
- New error cases:
  - Bind operation failed
  - Request timed out
  - Argument "too large" can occur if, e.g., a table grows
- Costs may be very high
- ... so RPC is actually not very transparent!

# RPC costs in case of local destination process

- Often, the destination is right on the caller's machine!
  - Caller builds message
  - Issues send system call, blocks, context switch
  - Message copied into kernel, then out to destination
  - Destination is blocked... wake it up, context switch
  - Destination computes result
  - Entire sequence repeated in reverse direction
  - If scheduler is a process, context switch 6 times!

# Synchronous vs. Asynchronous Communication

- The IPC operations may provide the synchronization necessary using *blocking*. A blocking operation issued by a process will block further processing of the process until the operation is fulfilled.
- Alternatively, IPC operations may be asynchronous or *nonblocking*. An asynchronous operation issued by a process will not block further processing of the process. Instead, the process is free to proceed with its processing, and may optionally be notified by the system when the operation is fulfilled.

# Using threads for asynchronous IPC

- When using an IPC programming interface, it is important to note whether the operations are synchronous or asynchronous.
- If only blocking operation is provided for send and/or receive, then it is the programmer's responsibility to using child processes or threads if asynchronous operations are desired.